CSE 347 Lecture 21

On-line Algorithms I: Amortized Analysis

1. Array-based stacks
2. Union-find

In average-case analysis, we saw that by averaging over all possible distinct inputs, simple algorithms could achieve excellent (average-case) performance. There is another way of averaging across inputs that makes sense for data structures in particular: we assume that a series of inputs or operations that we must handle arrive one at a time. This is an <u>on-line</u> input model. Now, when we use a data structure to implement an algorithm, we don't care so much how long each individual operation takes. Instead, we care about how long the operations take in aggregate — how many steps did accessing the data structure take during the entire execution of the algorithm? A convenient way of analyzing this is to consider how much time is taken per operation <u>on average</u> across the sequence. This is useful for data structures that are typically fast, but occassionally pay a large penalty — say the first 100 operations take 1 step each, but the 101st takes 100 steps (say to copy the 1st 100

elements) Then the amortized cost per operation is $\frac{\#\text{steps}}{\#\text{operations}} = \frac{200}{101} < 2$. As long as the expensive steps happen relatively infrequently for any sequence, we'd be happy with such a data structure.

Array-based stacks

As a simple example, let's consider implementing a stack using an array. Recall that a stack has two operations, push(x) that adds x to the "top" of the stack ("above" whatever was there previously), and pop, that removes the element from the top of the stack (returning the previous top element to the top position) and returning this removed element. It's easy to implement a stack with a linked list with O(1) time push & pop operations, so why bother? Locality might be a reason. So here is an alternative: we allocate an array of length ℓ, and have a "top" index, initially 0. (Pop returns NULL when top = 0.) Push(x) places x in index top, and increments top; pop decrements top (if top > 0) and returns the element in index top. What's wrong with this so far? Once top = ℓ, after we push, the array is full; another push will give an out of bounds memory access. So, if we reach top = $\ell+1$, a push operation allocates an array of size 2ℓ, and copies the contents of the old array to the new one. These ℓ copies cost $O(\ell)$ time, which seems bad. But, it can only happen after a large number of push

operations, so we'll be able to bound the amortized steps-per-operation.

A convenient way of analyzing such data structures is to assign the data structure some <u>potential value</u> $\Phi_t \geq 0$ after each tth operation. Now, suppose that initially $\Phi_0 = 0$ and if the tth operation takes T_t time, $T_t + (\Phi_t - \Phi_{t-1}) \leq c$ for some fixed c. Then the total time over n operations satisfies

$$\sum_{t=1}^{n} T_t \leq \sum_{t=1}^{n} c - (\Phi_t - \Phi_{t-1}) = nc - \underbrace{\Phi_n}_{\leq 0} + \underbrace{\Phi_0}_{=0} \leq n \cdot c$$

ie., the amortized time per operation is at most c.

Think of Φ_t as the state of a "bank account" — we can afford an expensive operation T_t as long as the amount we withdraw, $(\Phi_{t-1} - \Phi_t)$, is enough that we don't exceed our allowance of c dollars per operation, i.e., $T_t - (\Phi_{t-1} - \Phi_t) \leq c$.

So here, suppose that on each push operation when the array is not full, we spend \$1 to update top & store x, and deposit \$2 in the bank account. We also spend \$1 to check top>0, decrement it, and return the element on each pop — so both of these can be done with an allowance of \$3 per operation. Now, when we resized the array to size ℓ, we had $\ell/2$ elements in it. So the array only needs to be resized again after $\ell/2$ push operations. But, we deposited \$2 for

each of these operations, so our account has at least ℓ in it, which we can withdraw to pay \$1 to copy each of the ℓ elements. Thus, we take amortized time $O(1)$ per operation.

Union-Find

Now we'll analyze a data structure we needed to implement both Kruskal's Algorithm for minimum spanning tree, and Karger's Contraction Algorithm for global min-cut. For both algorithms, we needed to test if both endpoints of an edge were in the same set of vertices, or if they were in different sets. For Kruskal's algorithm, an edge would create a cycle if both endpoints were in the same set (in which case we discarded it), and otherwise we'd add the edge, combining two of the trees in our forest. For Karger's Algorithm? We needed to combine two sets of vertices and then delete the edges that had both endpoints in this new set. In summary, we needed to support the following three operations:

MakeSet(v): returns the "name" of a new set containing v.
Find({v}) returns the "name" of the set containing v.
Union(S_1, S_2): replaces the sets S_1 and S_2 with $S_1 \cup S_2$.

We'll give a data structure that supports MakeSet and Find operations in $O(1)$ time, and Union operations in amortized time $O(\log n)$.

We'll represent the sets by linked lists with an extra field that points directly to the head of the list in every node. We'll use the reference to the head as the "name" of the set, so given the node containing v, we can look up the "name" of the set containing it in O(1) time. We'll also include a size field that, in this head node, contains the size of the set. MakeSet(v) now only needs to initialize one of these nodes indicating that it is the head of its own list of size 1. This is O(1) time.

The interesting operation is Union(A, B). We'll compare the size fields, and use the "name" (head) of the larger of the two sets for the union A∪B. We update the size field of this node to |A|+|B|, and redirect its next field to the head of the smaller list. We then walk the smaller list, updating the head field to point to the new head. At the end of the smaller list, we update the final next field from NULL to the node that was formerly second in the larger list. We then return the head of the combined list—we do not touch any further nodes from the larger list.

Theorem n Union operations run in amortized time O(log n) each.

Proof: First, observe that after n union operations at most 2n distinct elements have been involved

in any union operations, so the largest set created has size at most $2n$.

Now, we wish to bound the total time we spend walking the lists. Notice, each time we walk the smaller of the two lists, so the size of the set containing these elements at least doubles. So, if the node is updated ℓ times, it is in a set of size at least 2^ℓ. But, since the size of the largest set is at most $2n$, $2^\ell \leq 2n$, and so $\ell \leq \log_2 n + 1$. Thus, every element is updated at most $\log_2 n + 1$ times, where each update takes time $O(1)$, so we take amortized time $O(\log n)$ overall. ▨

An alternative implementation of Union-Find obtains $O(1)$-time union operations at the cost of increasing Find operations to $O(\log^* n)$, where $\log^* n$ is the # of times we need to apply $\log_2 n$ to get 1. (Actually, an even smaller amortized time bound is possible, small enough that for all practical purposes it is constant.) We obtain this improved time bound by (1) lazily updating the name fields during a Find operation, (2) using a tree (with pointers from children to parents) instead of a list with separate next and head fields, and (3) using the <u>rank</u> of the tree rather than the size of the set to determine which should be the

root (name). The rank of a tree is recursively defined as the maximum over the ranks of its subtrees if there is a unique subtree of this rank; and otherwise it is one greater; leaves have rank 0. Note that this measures the height of the tree before we update the upward references during a Find.